Creating Telecommunication Services based on Object-Oriented Frameworks and SDL

Dr Richard Sinnott
GMD Fokus
Kaiserin-Augusta-Allee 31
Berlin, Germany
sinnott@fokus.gmd.de

Mario Kolberg
Dept. of Electronic and Electrical Engineering
University of Strathclyde
Glasgow, Scotland
mkolberg@comms.eee.strath.ac.uk

Abstract

This paper describes the tools and techniques being applied in the TINA Open Service Creation Architecture (TOSCA) project to develop object-oriented models of distributed telecommunication services in SDL. The paper also describes the way in which Tree and Tabular Combined Notation (TTCN) test cases are derived from these models and subsequently executed against the CORBA-based implementations of these services through a TTCN/CORBA gateway.

1. Introduction

The aim of the TOSCA project is to develop a service creation environment that enables multimedia-based telecommunication services to be produced in an effective manner, i.e. they are created rapidly but not at the expense of their reliability [18]. Central to the approach is that the services to be generated are validated. This validation is required both when the service is initially created and also when it is deployed in an environment where it may interwork with other services causing potentially undesired behaviours of one or more of the services. In this paper we focus predominantly upon the validation of isolated services. Aspects of service interworking and the issues involved in their validation are discussed in [6].

Validation of services implies that formality is introduced into the service creation process. Producing formal specifications of the system to be developed is a traditional starting point in applying formal techniques [13]. Unfortunately, it is often the case that formal techniques are used *only* at this stage of the software development process. Ideally, formality should be taken through to the final implementation of the software itself. This is a notoriously difficult activity often - depending upon the nature of the formal language and method - requiring arduous refinement and obligatory proof steps [23]. An alternative process to refinement of specifications through to implementations is to develop the specification and implementation as dual, i.e. concurrent, activities. Provided that the specification and implementation are at the same level of abstraction, the specification can be used as a basis for testing the implementation.

Specifications and implementations are typically not at the same level of abstraction however. Abstraction may be considered as a two-sided sword. On the one hand it enables simpler models of systems to be constructed and can thus act as a better requirements capturing technique. On the other hand the gap between the model of the software and the final software itself is increased. This reason is often cited by formal methods detractors who fail to see the advantage in formal models that bear little or no relation to the software under development and that all but a few specialists understand.

Distributed system development offers one area where the parallels between the development of specification and implementation can be more readily drawn, i.e. they can be expressed at the same levels of abstraction. Interface definition languages (IDL) when used as a common vocabulary for describing the syntactic aspects of interface interactions, serve as an ideal starting point for developing both specifications and implementations. Support of an IDL mapping is thus critical if a formal language is to be used to model distributed systems.

Few formal techniques have adopted current techniques in software development as has the ITU-T Specification and Description Language (SDL) [9]. As well as providing an IDL mapping which we describe in section 4, it offers state of the art tool support that few other formal techniques can match[1]. We discuss some of these tools and how they are being applied in TOSCA to develop, validate and derive tests from the SDL models in sections 4, 5 and 6.

IDL is only a basis for the development of both implementations and specifications. Given that rapid development of high quality services is a fundamental feature of service creation in TOSCA, developing specifications (and implementations) from nothing, or from an IDL only basis, is not a viable option. Instead techniques

[1] Lack of tool support is another common objection raised against formal techniques!

that can expedite the software development process are thus necessary. Whilst it is typically the case that implementations rarely (if ever!) start from nothing, the same cannot be said for the development of formal specifications. In TOSCA we are addressing this issue through the adoption of techniques based upon object-oriented implementation and specification frameworks. We note that these frameworks are developed both in the implementation world, e.g. using C++ and distributed technologies such as CORBA [3] and the specification world, e.g. using SDL. Our focus in this paper is based upon the development of SDL frameworks, their usage in developing real services and how these models of services can be applied to test the real implementations of those services.

2 The TOSCA Approach to Service Creation

The TOSCA project proposes an approach to service creation which should provide both for rapid service provisioning and for high service quality. The approach assumes that for certain categories of service, a flexible and reliable software ***framework*** is developed. The concept of framework based software engineering has arisen to help to realise the holy grail of software engineering: *re-use*. Frameworks are a natural extension of object-oriented techniques [7]. Whilst object technology provides a basis for re-use of code, it does not provide features to capture the design experience as such. Frameworks have developed to fulfil this need.

A framework can be regarded as a collection of pieces of software or specification fragments that have been developed to produce software of a certain type or niche [5]. A framework is only partially complete. Typically, they are developed so that they have holes or flexibility points in them where service specific information is to be inserted. This filling in (*specialisation*) of the flexibility points is used to develop a multitude of services with differing characteristics.

In TOSCA, this specialisation may be done by non-technical people, e.g. business consultants, through paradigm tools. Paradigm tools offer a graphical and intuitive means whereby services can be designed. Thus the service designer should not necessarily have to consider the lower level behaviour of the service to be able to create one. Rather, they should be provided with a high-level representation of the service components and the ability to *tune* their behaviour and how they are composed with one another. In this paper we do not focus on the issues in the application of paradigm tools to specialise implementation-oriented and specification-oriented frameworks to create service implementations and service specifications respectively. More information on the paradigm based approach to service development can be found in [11].

Once the SDL framework has been specialised to create a completed model of the service, in the first instance, it is necessary to provide some immediate feedback to ensure that the service behaviour is as desired. This is achieved through (graphically) animating the service behaviour. Once the basic functionality of the service is satisfactory to the service designer, a more detailed check on its behaviour is required, e.g. to ensure that it has certain properties such as deadlock/livelock freedom. Once the specification has been thoroughly checked it is then used to derive test suites against which conformance of the service implementation to the specification can be tested.

The immediate question that arises from the TOSCA approach is where do the frameworks come from? Given the nature of the services to be created within TOSCA, we have based our services around the Telecommunications Information Networking Architecture (TINA), or more specifically the Service Architecture [15] and Network Resource Architecture [16] of TINA and an existing implementation of a multimedia-conferencing service based on TINA [19].

3 Development of Frameworks based on TINA

The TINA architecture introduces the underlying concepts and provides information on how telecommunication applications and the components they are built from, have to behave. Central to the architecture is the concept of a *session*. Three sessions are identified:

- **access session**: this represents mechanisms to support access to services (service sessions) that have been subscribed to.
- **service session**: includes the functionality to execute and control and manage sessions, i.e. it allows control of the communication session.
- **communication session**: controls communication and network resources required to establish end to end connections.

Currently, the service session has been the main area upon which frameworks are being developed in TOSCA. The relation between the sessions is depicted in Figure 1.

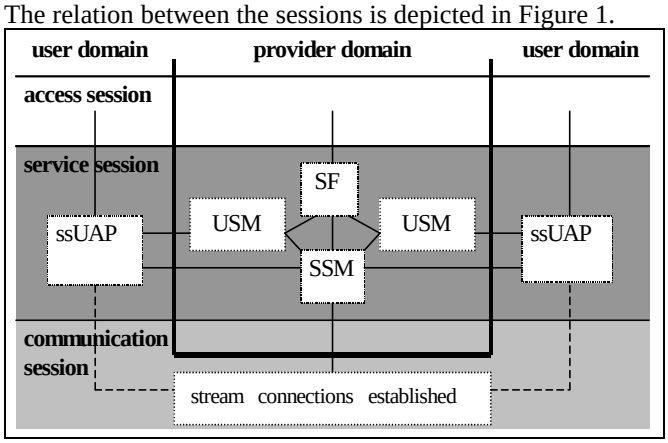

Figure 1: Relation Between the TINA Sessions

Here the service session user application (ssUAP) represents the users interface to the service, i.e. it determines how they may participate in the service. The Service Factory (SF) is used to create instances of services when requested to do so by components in the access session: namely user agents. Broadly speaking, an instance of a service typically consists a Service Session Manager (SSM) to control the global service behaviour, and a collection of User Service Session Managers (USM) – one of each is used to control a users participation in the service.

Typically, users can join services, suspend, resume or terminate their participation in services. The logic associated with these requests are processed in the service session, e.g. whether the user is able to resume themselves in the service at that time. If successful, the appropriate connection operations are invoked on the communication session, e.g. resume my previously suspended connections.

It is important to note that this architecture does not overly constrain the kinds of services that can be created from it. Rather, it acts as a template for a multitude of services, e.g. multimedia conferencing services, chatline services, videophone services, neighbourhood watch services or newsflash services. Indeed even within these services there exist a plethora of variations. In multimedia conferencing for example, there might be differing roles, e.g. chairman, observer, participant. These differing roles might result in differing expected functionalities, e.g. only chairman can invite (or suspend or terminate) other users, only participants can vote. Users might be able to have differing charging (or billing or accounting) possibilities, e.g. reverse or split charging, or other variations.

As well as these role specific specialisations, numerous others are possible also, e.g. only start the service if a certain number of successful responses to the invite have been received, or quit the service if the number of users falls below a certain level (or if the total charges generated from using the service falls below a certain level). It is precisely these variations on the general theme that specialisations are expected to capture whilst the general theme itself is represented by the framework.

To engineer frameworks it is thus necessary to have a core behaviour. In TOSCA this core behaviour is based around the informal (textual) description of the behaviour of the service session components; their Object Definition Language (ODL) [17] description and parts of the ERI-TIMMAP multimedia videoconference service [19]. TINA ODL is a superset of IDL which allows amongst other things, to distinguish between *supported* and *required* interfaces. ODL also allows for the expression of *groups* of objects and the objects used to manage those groups.

4 Overview of the TOSCA Framework

If developing software is a complex activity then developing frameworks is more complex again. Developing a framework so it removes large parts of the problem of service design, thus expediting the creation process, whilst still offering a means to create numerous different kinds of services is an especially challenging activity. To produce successful frameworks requires that the points where design decisions are made, are made flexibility points. Using frameworks to produce services then requires that these flexibility points are made available so that new design choices can be taken to produce new services or service flavours. Perhaps the hardest part of the framework development process is the identification of these flexibility points [10].

In TOSCA we focused on a small set of flexibility points. This set of flexibility points allowed us to produce a multitude of different services with different types of behaviour. Specifically, we chose the following flexibility points:

- the start up, suspension, resumption and termination of users sessions;
- the start up, suspension, resumption and termination of service sessions.

In producing a framework it is necessary to have fixed places where the flexibility points are to exist. Flexibility points cannot simply be placed anywhere in the design of the framework. Rather, flexibility points may only be filled in (specialised) at certain fixed times. Thus it is necessary to represent the points of flexibility directly in the design of the framework, but the actual behaviour associated with these flexibility points is effectively NULL until they are specialised. To achieve this we introduced appropriate IDL operations that were associated with the appropriate objects in the framework design. As an example we consider the USM and the representation of the flexibility points concerned with the start up, suspension, resumption and termination of users sessions.

The simplified ODL for the USM modelled in the TOSCA framework is

```
group USM
{components UFS, ...;
 manager UFSmgr;
 contracts i_UFSmgr,i_Callback, i_ControlWindowHandler
           ...; };
```

The UFSmgr object is responsible for controlling the objects in the specialisable part of the USM. In reality this means that it should – amongst other things - be able to: terminate, suspend or resume existing all objects it controls, or add new (named) objects to those it currently knows about[3]. This implies that all objects controlled by the UFSmgr support this basic *lifecycle* functionality, i.e. upon reception of certain signals all objects can suspend, resume

[3] Typically, the object name and a null reference are passed in and the name and PId of the created object returned.

or terminate themselves. To achieve this, all objects inherit from interface i_CO_lifecycle. The IDL for this interface is:

```
interface i_CO_lifecycle
{ void initialiseObject(in PropertyList initInfo,
                        in Object mgrRef);
  void suspendObject();
  void resumeObject(in Object mgrRef);
  void terminateObject();
  ... };
```

When initialised or resumed, objects need to be made aware of the reference for their managers. This allows for later checks on arriving invocations, i.e. to check that they originated from their manager. As well as supporting this core functionality, the interface to the UFSmgr (i_UFSmgr) supports other operations. The default behaviour for the UFSmgr is that it allows a user to suspend and terminate their participation in the current session. The IDL for the i_UFSmgr interface is:

```
interface i_UFSmgr : i_CO_lifecycle
{void suspendSessionRequest();
      // called by user to suspend their session
 void terminateSessionRequest();
      // called by user to terminate their session
 void suspendAll();
       // used to suspend USM and all associated objects
 void requestObject(inout NamedObject obj);
      // called to  create window handlers
 oneway void ufsstart();
      // not implemented in framework – specialised!
 oneway void ufssuspend();        //    "
 oneway void ufsresume();         //    "
 oneway void ufsstop();           //    "
 ... };
```

We point out here that we define several operations whose sole purpose is to act as a placeholder in the framework design through which the specialisation can be achieved, i.e. these represent the points of flexibility that enable us to have different service behaviours. The other behaviours may be implemented directly.

In terms of formal specification development the first stage in this implementation step is to map the ODL descriptions to the formal language of choice. We note here that relatively few IDL to formal specification language mappings have been made. One of the reasons for this is the lack of support for object-orientation in the formal specification world and more particularly the lack of support for object references as first class entities, i.e. they can be passed around as parameters. SDL is one of the few languages that support a mapping. In TOSCA the Y.SCE tool [21] was used to import the ODL/IDL description and generate the associated SDL stubs and skeletons. The following table summarises some of the main features of the ODL/IDL to SDL mapping used (and implemented) in TOSCA.

ODL Structure	SDL Mapping
Group type	block type
Object type	block type
Interface type	process type
Object Reference	PId
Oneway Operation	signal prefixed with pCALL_
Operation	signal pair prefixed with pCALL_ and pREPLY or pRAISE resp.
Exception	signal prefixed with pRAISE_
Basic IDL types	syntype
any	not supported
enum	newtype with literals
typedef	syntype
struct	newtype with associated structure
constant	synonym

Table 1: Summary of ODL/IDL to SDL Mapping

Other mappings have been made from IDL to SDL [1], however these are based largely around the remote procedure call concept of SDL. The remote procedure concept in SDL is a shorthand notation and is based on a substitution model using signals and states. More precisely, remote procedures are decomposed into two signals. The first carries the outgoing parameters (**in** or **inout**) and the second the return value of the procedure and all **inout** parameters. These signals are sent via implicit channels and signalroutes. There are several problems with mapping IDL operations to remote procedures. For example, they prohibit the raising of exceptions – an essential feature in realistic distributed systems. Also, the client side of the remote procedure call is blocked until the server side returns.

One point worth noting here regarding the mapping is the modelling of object types through blocks in SDL. In SDL, it is not possible to create blocks dynamically, although this is one extension to the language that may well be incorporated in the next version of SDL (SDL-2000). It is possible to model SDL systems where the perception of object (block) creation is achieved. One way of achieving this is through providing process instances that exist at start-up time whose sole purpose is to create other processes in that block when requested. It is this approach we adopted in TOSCA. These creator processes supported a single exported remote procedure (which could be imported into other blocks and subsequently called) operation that created an instance of the manager process, e.g. the UFSmgr given previously. References to this manager process, i.e. the process identifier, would then be returned to the requesting object.

As with other IDL language mappings, client stubs and server skeletons are generated. These act as templates whose behaviour is to be filled in through inheritance. These stubs and skeletons are placed in two SDL packages: *Name_Interface* and *Name_Definition*. The *Name_Interface* package contains the interface specifications in the form of

data types, signals, remote procedures, signallists etc. An example of the contents of the Name_Interface package is given in Figure 2.

```
Package Name_Interface
signal pCALL_i_UFSmgr_suspendSessionRequest;
signal pCALL_i_UFSmgr_terminateSessionRequest;
signal pCALL_i_UFSmgr_ufsstart; // and ufsstop, ufsresume, ufssuspend
signal pCALL_i_UFSmgr_suspendAll;
signal pCALL_i_UFSmgr_requestObject(NamedObject);
// and associated pREPLY_ signals – but not for oneway operations
signallist i_UFSmgr_INVOCATIONS =
 pCALL_i_UFSmgr_suspendSessionRequest,
 pCALL_i_UFSmgr_terminateSessionRequest,
 pCALL_i_UFSmgr_ufsstart, // and ufsstop, ufsresume, ufssuspend
 pCALL_i_UFSmgr_suspendAll, pCALL_i_UFSmgr_requestObject;
signallist i_UFSmgr_TERMINATIONS =
  pREPLY_i_UFSmgr_suspendSessionRequest,
  pREPLY_i_UFSmgr_terminateSessionRequest,
  pREPLY_i_UFSmgr_suspendAll, pREPLY_i_UFSmgr_requestObject... ;
```

Figure 2: Example of Name_Interface Package Contents

This package is then *used* in the definition of the Name_Definition package. Figure 3 gives an example of the kind of SDL generated focusing on the i_UFSmgr interface of the UFSmgr object:

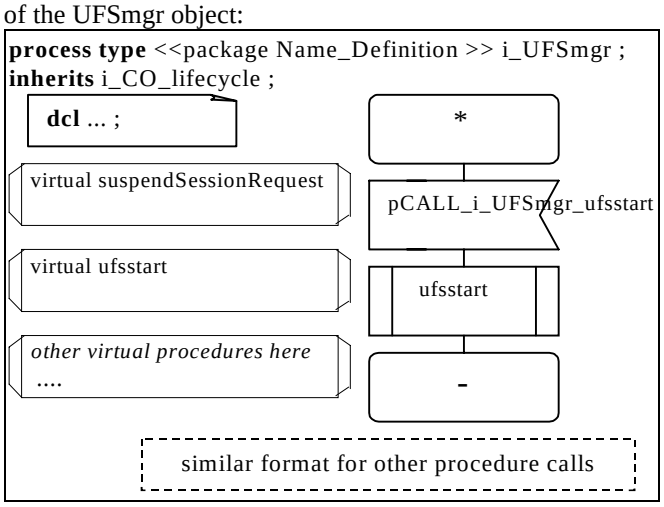

Figure 3: Example of Name_Definition Package Contents

The virtual procedure for the ufsstart (and all oneway operations) consist of a virtual start transition followed by an immediate exit. In non-oneway operations, the generated procedures contain a pREPLY_ signal of the appropriate kind. Along with the virtual procedure definitions, signals and (asterisk) states are also generated that result in the procedures being called.

As an example of the way in which the generated SDL server skeletons can have their core behaviour inserted, i.e. the behaviour before they are specialised, we consider the implementation of the i_UFSmgr interface (i_UFSmgrImp) of the UFS object given previously. The default behaviour for the UFSmgr is that it creates a control window handler only. A simplified example of the structure of this object is given in figure 4.

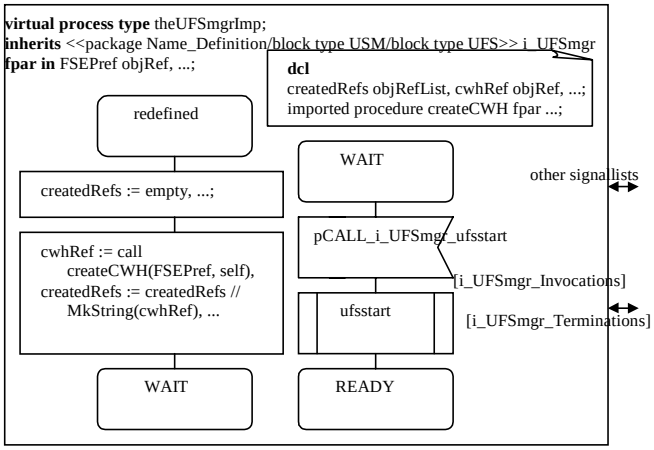

Figure 4: Structure of Basic UFSmgr

This process type is parameterised with (amongst other things) the reference to the user application. When an instance of this process type is created, initialisation of local variables is done, e.g. the list of created references is set to empty, and the default behaviour of creating a control window handler is made. As discussed, this requires that the necessary exported remote procedure is imported. Following this default behaviour, the UFSmgr is ready to be specialised, i.e. it is in a state where it can accept signal pCALL_i_UFSmgr_ufsstart.

As stated, the specialisable procedures have null behaviours, i.e. start and exit. This allows for the behaviour of the framework as a whole to be checked without necessarily having any specialisation taking place, e.g. the basic USM behaviour (and SSM and SF) behaviours can be checked to ensure the framework as a whole correctly represents the informal (textual) requirements. Once the core behaviour has been specified and verified, the framework can be saved as a package and *used* in defining services, i.e. SDL systems.

5 Specialising the Framework to a Service

As an example of framework specialisation we focus on a videophone service here. A videophone service is mainly characterised by having two different kinds of users, a caller and callee who have an audio-visual connection that allows them to see and speak to one another. There may only be one instance of each of these users in the session at one time. In TOSCA, a “kind” of user is represented by a user role which has a set of privileges and characteristics attached to it. Each member in a session is assigned a role on joining the session and may hence only perform the corresponding activities. In the case of the videophone service, the invoking member, i.e. the user who starts the service is automatically assigned the role of caller. The distinction between the caller and callee is that the caller is able to invite users, i.e. callees to join the service. When the caller terminates the whole service session terminates.

When a callee terminates, the caller may issue other invitations, i.e. the service does not terminate when the callee quits their participation in the session. As discussed previously, the framework already caters for session members being able to suspend and resume their participation in the session. However, in this particular videophone service we wish to have the additional restriction that a suspended callee only has thirty seconds to resume their session, otherwise they are automatically quit from the session. The ssUAP associated with the caller is shown in Figure 5. The callee's ssUAP is similar except that it does not support invitations.

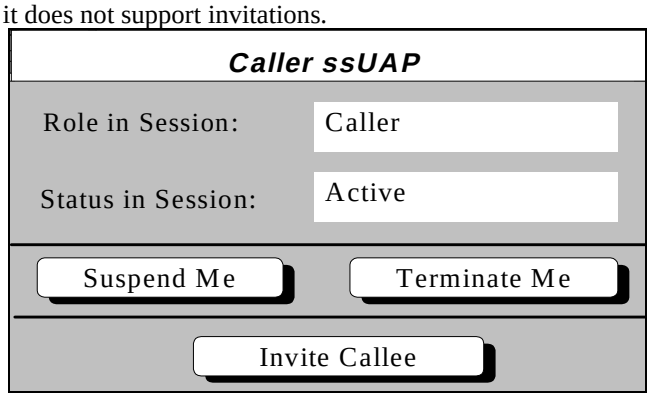

Figure 5: The ssUAP Associated with the Caller

To achieve this service specialisation we *use* the SDL package representing the framework. Both simple and virtual inheritance are used to specialise the components in the framework. Simple inheritance is used at the upper block level, e.g. the USM block level. Subsequent block types, e.g. the UFS block type as well as process types and procedures are reused by virtual inheritance. Hence the UFSmgrImp process type given above is declared as virtual. This is necessary since virtual inheritance allows the communication links, i.e. channels and signalroutes in the framework to be reused (and possibly extended). Virtual inheritance does not, however, allow for multiple redefinitions in one scope (e.g. different types of USM at system level, for caller and callee). As a result, it is not possible to use virtual inheritance for the top-level block types: simple inheritance is used instead.

Creating the roles, caller and callee requires that the USM block type in the framework is specialised to two different block types: callerUSM and calleeUSM respectively. The ufsstop procedure of the caller is specialised so that when invoked a signal is sent to terminate the whole service session, i.e. when the caller quits the service is terminated.

To realise the cardinality constraints explained above, the SSM needs to be specialised as well. The SSM knows about all members and their current state in the session and is responsible for allowing new users to join the session. The SSM for this videophone service is thus specialised to restrict the maximum cardinality of callers and callees to 1 and the minimum cardinality of callers and callees to 1 and 0 respectively.

As stated previously, the operation ufsstart in a UFSmgr is executed whenever a new instance of the USM is created, i.e. a user has joined the session. For the caller role the specialised ufsstart procedure contains a call to create an Invitation Window Handler. The specialisation of the caller's ufsstart procedure is represented in figure 6.

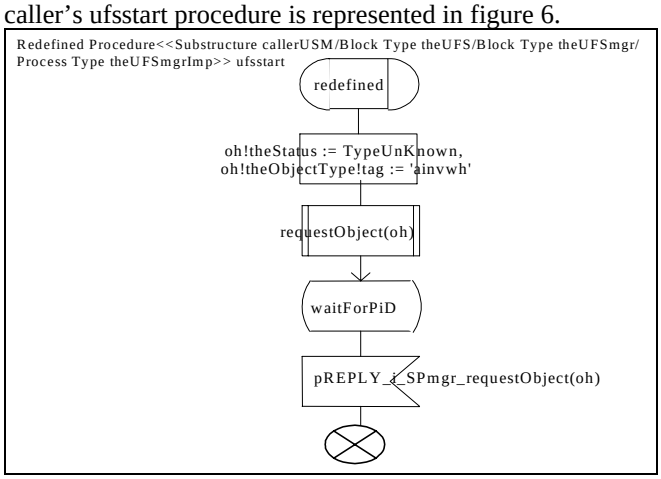

Figure 6: Specialisation of the Caller's ufsstart procedure

The specialisation for the callee is twofold. Firstly, the audio-visual connection between caller and callee needs to be created and secondly, a timer needs to be introduced to make sure that a suspended callee is terminated from the session after 30 seconds. For setting up the connections, the callee's ufsstart procedure is specialised so that the SSM is firstly queried to find the users active in the session, i.e. get the information on the caller, and then a call is made to the SSM to make the appropriate connections on the communication session between the caller and the callee.

To ensure that a suspended callee is terminated after thirty seconds, a *timer* is introduced in the callee's UFSmgrImp. This timer is set when the user suspends their session. It is reset when the callee resumes their session. If the timer times out then a signal is sent to the UFSmgr of the callee with a request to quit the session. The specialisation of the callee's ufssuspend and ufsresume are shown in Figure 7.

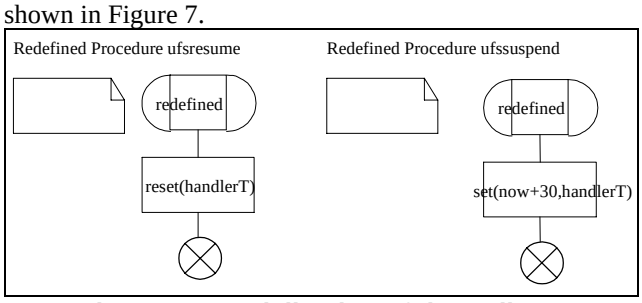

Figure 7: Specialisation of the Callee's Suspend and Resume Flexibility Points

The specialisation of the callee's UFSmgrImp when it is in state suspended so that it takes some action should a timeout occur is given as:

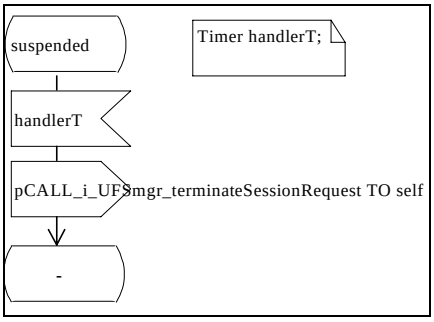

Figure 8: Specialisation of the Callee's UFSmgr

It is important to note here that the actual calling of the ufsstart, ufsstop, ufsresume and ufssuspend procedures always occurs at fixed points in the overall behaviour of the framework. That is, ufsstop, ufsuspend, ufsresume are only called when a valid terminate, suspend or resume request respectively are received by the UFSmgrImp when it is in state ready/suspended, ready or suspended respectively.

Once the framework has been specialised, in the first instance, it is animated to give the service creator feedback on its functionality. As well as the user interface creation being animated, e.g. new windows are created on the ssUAP, we have focused - amongst other things - on producing graphical animations of the interface to the communication session, e.g. showing the connections between users in the session and how they are modified when new users join, or existing users suspend or resume their participation in sessions. It is important to note that the objects performing the animation, i.e. the GUIs, are themselves CORBA objects. A detailed discussion on the animation activity can be found in [11].

6 Developing Test Cases from the SDL Service Model

Once the service has been exhaustively animated and has all expected properties validated, it may then be used to generate test cases to ensure that the implementation has the correct functionality, i.e. that it is conformant with the specification. Conformance testing is an especially challenging area in the formal methods community. Since systems, especially those based on TINA session models, may be very complex, it is impossible to completely test all possible behaviours of a system. Instead, testing is normally done by identifying certain important or essential tests of a given system - this is an especially challenging task given the distributed nature of the services in TOSCA.

Since the specification and implementation should be based on the same IDL, the nature of the tests should not have to change. For example, an isolated test of the specification is likely to involve interacting at an operation in an interface of a given computational object and ensuring that when invoked the correct response is eventually received. The same test should be applicable to the implementation assuming that they are based on the same IDL, i.e. the interface and accompanying operation name will be the same and the parameters should be the same. Typically, such simple tests are not the normal scenarios in distributed systems, where isolated invocations on a given interface require other remote interfaces to be monitored (so called points of control and observation - PCOs) to ensure that the invocation was as planned. The IDL basis for the testing and observation is the same though.

It is quite possible that given tests can be represented directly in the specification language, i.e. in the form of SDL processes to test the behaviour of the specification, or MSCs. These MSCs can be used by implementors to ensure that when interpreted in C++ for example, the implementation allows for the same sequence of events as given in the MSC. However, instead of relying upon an interpretation of some notation, e.g. SDL or MSCs, a standardised testing language exists: the Tree and Tabular Combined Notation (TTCN) [20]. This can then be used to generate code to test the implementation.

With TTCN an Abstract Test Suite (ATS) is specified which is independent of test system, hardware and software. Test suites consist of a collections of test cases, where each test case typically consists of sending messages to the implementation under test (IUT) and observing the responses from the IUT until some form of verdict can be assigned, e.g. whether the result was a pass, fail or inconclusive. Matching may be done on the data structures of the received messages which may themselves be expressed either through Abstract Syntax Notation One (ASN.1) or native TTCN. The test cases themselves are represented in tabular form (TTCN.GR), although TTCN also occurs in machine processable form (TTCN.MP).

We note that the IUT itself is treated as a black box, i.e. only it's observable interfaces are considered. The points at which it is tested are termed points of control and observation (PCO). Once an ATS is complete it is converted to an Executable Test Suite (ETS) which can then be used by the system performing the test to see whether the implementation passes or fails the tests.

A TTCN specification has a standardised layout consisting of four major parts:

- **overview part** containing a table of contents and description of the test suite;
- **declarations part** declaring all messages, timers, variables, data structures, PCOs;
- **constraints part** assigning values and creating constraints to check the responses of the IUT;
- **dynamic part** containing all test cases, test steps, default tables and verdicts, i.e. it describes the actual execution behaviour of the test suite.

Several tools exist within the Telelogic TAU toolset [14] that allow for the derivation of tests from SDL specifications. The Autolink tool of the SDT Validator allows for the semi-automatic generation of TTCN test suites based on SDL specifications.

Development of test suites from the SDL models can also be made interactively through the SDT TTCN link tool. This tool provides an environment that links the SDL specification world represented by the Specification Design Tool (SDT) with the testing world represented by the Interactive TTCN Editor and eXecutor (ITEX) tool. Once a TTCN link executable is generated from the specification it may be opened with ITEX and used to generate the declarations used to test the system. In effect this corresponds to generating mappings for the SDL channel names, the signals they carry and the parameters associated with these signals that the specification has with its environment. The SDL channels are mapped to PCO type declarations, the signals are mapped to ASN.1 abstract service primitive (ASP) type definitions and signal parameters mapped to ASN.1 type definitions. An extra TTCN table is also generated called OtherwiseFail. This table is used to catch all other ASPs at the PCOs, i.e. signals on channels, other than those listed in the test case through an ?OTHERWISE statement. These result in a fail verdict for the test. This table also accepts arbitrary timeout signals which result in an inconclusive test through a ?TIMEOUT statement. This table is used as the default for the test suite.

Having generated the static parts of the tests, the dynamic parts and the constraint parts associated with the test case can be developed through synchronising the TTCN test case with the SDL system. Once synchronised, the messages to be sent and received can be selected, i.e. the PCOs used (channels to/from the specification) together with the ASN.1 ASPs they carry from the list of possible SDL signals at that time. Once a PCO and ASN.1 ASP has been selected the constraints associated with the signal, e.g. the values of the parameters being sent or the acceptable values that are being received, can be set.

As an example of one test of the videophone service developed earlier, we consider the termination of a callee's session when they have been suspended for more than thirty seconds, i.e. they have not issued a resume session request within thirty seconds of being suspended. To perform this test requires that we declare two timer objects as shown in figure 9.

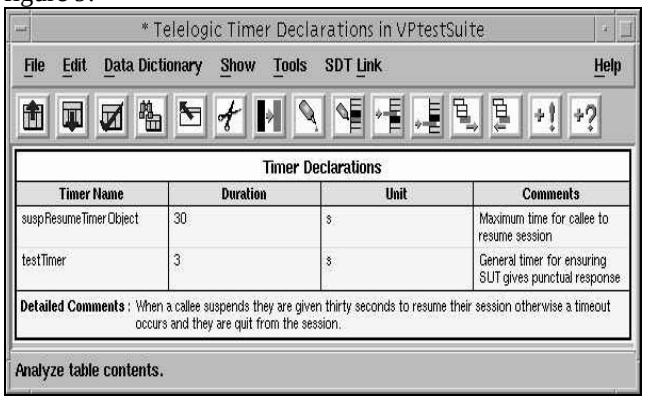

Figure 9: Timer Declarations in TTCN Test Suite

The timer declaration suspResumeTimerObject is used to check whether a resume request occurs within thirty seconds of the callee suspension. We note that we use seconds as the units of measurement but this may also be measured in values between picoseconds and minutes. We also note that the duration of timers may be overidden when they are started. In the successful resumption scenario, timer suspResumeTimerObject is parameterised by a test suite operation *rand(seed)* that generates a random value between zero and thirty. When this timeout occurs a signal to resume the callee's session is sent by the tester. In the unsuccessful case, this timer is started with the default value of thirty seconds. We introduce testTimer as a general purpose timer to ensure that responses are received within appropriate times from the SUT[5].

To run the test we require that the following test steps are carried out successfully:

- the caller starts the videophone service and joins in the session (1. in Figure 10);
- the caller sends out an invitation to a particular callee to join (2. in Figure 10);
- the callee receives the invitation and then joins in the service (3. in Figure 10);
- the callee then suspends their session (4. in Figure 10);.

For brevity we do not provide all of these test steps. We note that they all contain provisional pass verdicts (P) if successful.

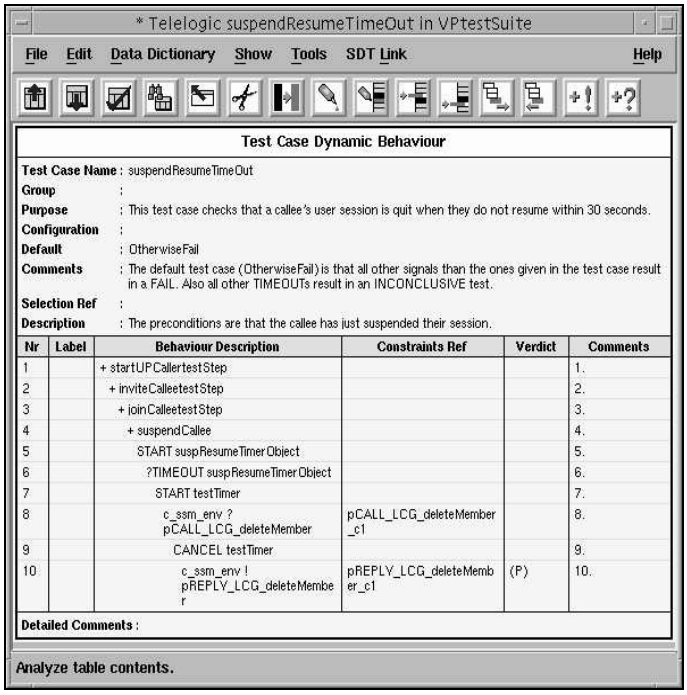

Figure 10: Test Case to Check the Termination of Prolonged Suspended Callee Sessions

[5] The default being three seconds.

We note that the OtherwiseFail is the default behaviour for this test case. Once a callee's session has been suspended (4.) timer suspResumeTimerObject is started (5.). This was declared to timeout after thirty seconds. Once this timeout occurs (6.) we set another timer (7.) which is used to ensure that the videophone service terminates the user session within a certain time period. As a default we declared the timer to have a duration of three seconds. The termination of the callee's session is indicated by the receipt of a call from the service session manager on the logical connection graph that exists in the TINA communication session (8.). Once this message is received successfully, e.g. the ?TIMEOUT statement in the OtherwiseFail table never occurred, the timer is cancelled (9.) and a send is issued stating that the callees connections have been deleted successfully by the logical connection graph. Once this message has been sent a final PASS verdict is assigned to the test case (10.).

For brevity we do not provide the postamble test steps associated with this test case, i.e. the termination of the caller session and quitting of the videophone service as a whole.

6.1 Execution of the Test Suites

Once the abstract test suites are completed, it is necessary to convert them into executable test suites. To achieve this requires that the executable test system is integrated into an environment where the SUT resides. In the case of the TOSCA services, this corresponds to having the ETS existing in a CORBA environment. The TTCN/CORBA gateway provides such an environment [22]. Figure 11 illustrates the structure of the gateway and its relation to the ETS.

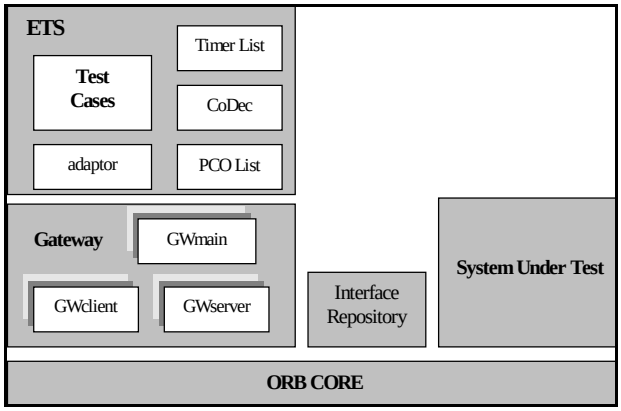

Figure 11: Structure of ETS and Gateway in Test Execution Scenario

The ITEX tool serves as the basis for developing the ETS from the ATS. This tool allows for the translation of the TTCN ATS into C-code. In addition to this it also provides an environment – the Generic Compiler Interface (GCI) - whereby the test cases can be adapted to the particulars of the implementation. In reality, the GCI requires that certain operations are implemented, e.g. coding functions and the interpretation of PCOs and timers [22].

The actual gateway itself consists of a main part and client and server parts. The main part is used for creating instances of the client and server parts and acting as the interface to the ETS. The client parts are used when test under consideration is on one of the supported interfaces [17] of the SUT, i.e. the SUT is acting as a server. The server parts of the gateway are used when the SUT is acting as a client, i.e. the SUT makes an invocation on a required interface [17] (represented as the GWserver). We note that the interactions between the gateway and the ORB core are made through the dynamic invocation interface (DII) for the gateway acting as a client of the SUT and through the Dynamic Skeleton Interface (DSI) for the gateway acting as a server of the SUT. Having run time knowledge of the types for the DII and DSI operations is achieved through the interface repository. As discussed in section 4, the IDL basis for the model (and hence test cases derived from the model) and implementation are of course the same.

7 Conclusions

This paper has tried to give a flavour of the tools and techniques that are currently being applied in TOSCA to develop telecommunication services based around the TINA architecture. We have seen how it is possible to take a semi-formal description given in TINA ODL, CORBA IDL and informal text to develop an object-oriented framework in SDL. We have also highlighted how this framework can be specialised to create instances of services. We produced an instance of a videophone service with explicit temporal constraints on user suspension and resumption. Having an SDL model allows us to investigate more deeply the behaviour of the service, e.g. through simulations and applying state space exploration tools. Once the service developer is satisfied with the overall functionality of the developed service, i.e. it possesses all properties that they desired, it may then be used to generate test cases against which conformance of the real implementation to the specification can be checked. These test cases are executed through a TTCN/CORBA gateway. Through this we hope that we have shown that the TOSCA approach allows formality to be taken right through the whole software development lifecycle.

Currently, the work in TOSCA has regarded the service specification and implementation development as dual (concurrent) activities. Whilst possible, this does incur more work, i.e. two frameworks have to be created and conformance of the services generated from them has to be established. To reduce the overall workload in framework and service development, ideally the SDL model should be used to develop the service implementation directly. Work is thus ongoing to assess the feasibility of using SDL as an implementation prototyping language where code

generation techniques are adapted to reflect the structure of the specification, i.e. partitioned code is generated that reflects the structuring and behaviour of the CORBA objects modelled.

More information about the current status of the work in TOSCA can be found at: http://www.teltec.dcu.ie/tosca/

8 References

[1] M. Björkander, Mapping IDL to SDL, Telelogic AB, 1997.

[2] M. Born, A. Hoffmann, M. Winkler, J. Fischer, N. Fischbeck, *Towards a Behavioural Description of ODL*, Proceedings of TINA 97 Conference, Chile.

[3] *The Common Object Request Broker Architecture and Specification*: Revision 2.0, Object Management Group, Inc., Framingham MA., July 1995.

[4] J.A. Hall, *The Seven Myths of Formal Methods*, IEEE Software, volume 7(5), pages 11-19, September 1990.

[5] R. Johnson and V. Russo, *Reusing Object-Oriented Designs*,Urbana, Ill., May 1991.

[6] M. Kolberg and E. Magill: *Service and Feature Interactions in TINA*, Proceedings of Feature Interaction Workshop'98, Lund, Sweden 1998.

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall 1991.

[8] I. Schieferdecker, M. Li, A. Hoffmann, *Conformance Testing of TINA Service Components - the TTCN/CORBA Gateway*, Proceedings of the Intelligence in Networks & Services Conference 1998, Antwerp, May 1998.

[9] International Consultative Committee on Telegraphy and Telephony - *SDL - Specification and Description Language*, CCITT Z.100, International Telecommunications Union, Geneva, Switzerland, 1992.

[10] R. Sinnott, *Frameworks: The Future of Formal Software Development*, Journal of Computer Standards and Interfaces Journal, special edition on Semantics of Specifications, August 1998.

[11] R. Sinnott, M. Kolberg, *Business-Oriented Development of Telecommunication Services with SDL*, Proceedings of OOPSLA Workshop on Precise Behaviour Specifications of OO Systems and Business Specifications, Vancouver Canada, October 1998.

[12] R. Sinnott, M. Kolberg, *Engineering Telecommunication Services With SDL*, Proceedings of Conference on Formal Methods for Open, Object-based Distributed Systems, Florence, Italy, February 1999.

[13] R. Sinnott, *An Architecture Based Approach to Specifying Distributed Systems in LOTOS and Z*, PhD Thesis, University of Stirling, Scotland, June 1997.

[14] Telelogic AB, *Getting Started Part 1 - Tutorials on SDT Tools*, Telelogic AB, 1997.

[15] TINA-C, *Service Architecture*, version 5.0, 16 June 1997.

[16] TINA-C, *Network Resource Architecture*, Version 3.0, February 1997.

[17] TINA-C, TINA Object Definition Language MANUAL, version 2.3, July 1996.

[18] TOSCA Consortium Deliverable 6, *Initial Approaches to the Specification and Validation of TINA Services, Internal Deliverable* AC237/GMD/WP3/DS/R/009/a1.

[19] TOSCA Consortium, *Specification of the Service Session Framework Targetted at the Eri-TIMMAP Platform*, AC237/ETL/WP2-4/PI/I/036/A4.

[20] Information technology – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN), ISO/IEC 9646-3 1997 (E).

[21] For more information see web address: http://www.fokus.gmd.de/minos/y.sce.

[22] M. Li, *Testing Computational Interfaces of TINA Services Using TTCN and CORBA*, Diplomarbeit, Department of Electrical Engineering, Telecommunication Network Group, Technical University Berlin, 1997.

[23] *Basic Reference Model of ODP -Part 4: Architectural Semantics*, ISO/IEC International Standard 10746-4, ITU-T Recommendation X.904, Geneva, Switzerland 1997.